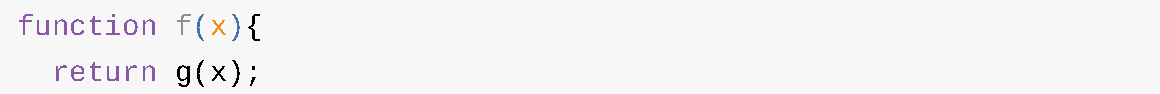
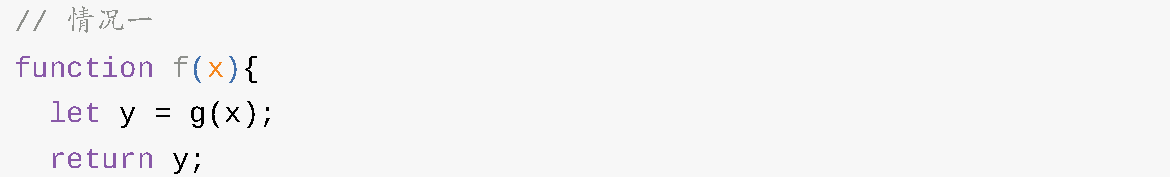
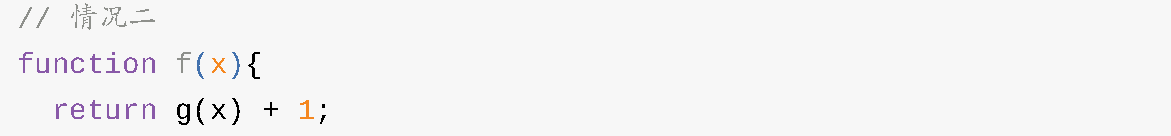
* + - 1. 尾调用优化
         1. 概念

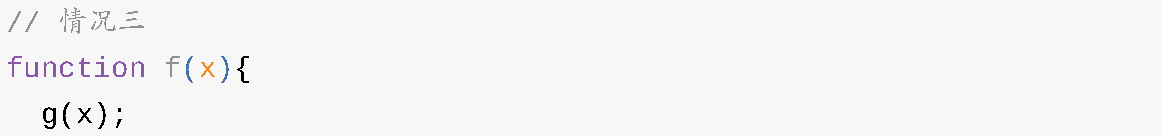
即函数的最后一步是调用另一个函数。



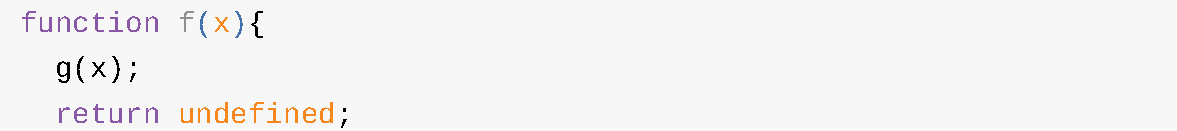
以下3种情况不是尾调用：



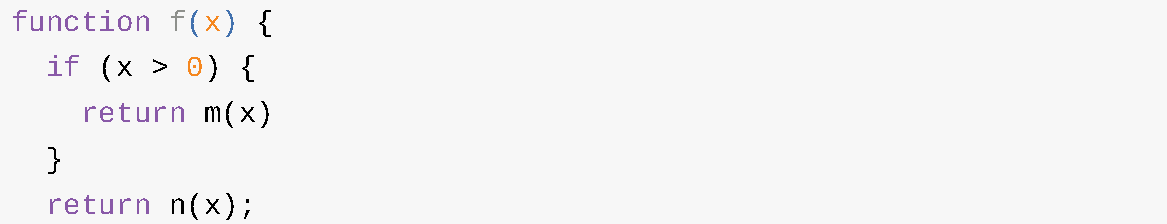




情况一是调用函数g之后，还有赋值操作，即使语义完全一样。情况二也属于调用后还有操作，即使写在一行内。情况三等同于下面的代码。



尾调用不一定出现在函数尾部，只要是最后一步操作即可。



* + - * 1. 尾调用优化

函数调用会在内存形成一个调用记录，又称调用帧，保存调用位置和内部变量等信息。如果在函数A内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用函数C，那就还有一个C的调用帧，以此类推。所有的调用帧，就形成一个调用栈。

尾调用因为是函数的最后一步操作，没有进一步的操作，不会再用到外层函数的变量，所以不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了。

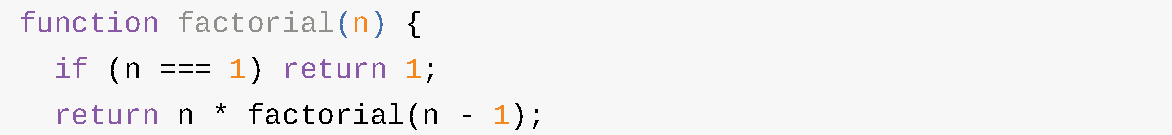
只有不再用到外层函数的内部变量，内层函数的调用帧才会取代外层函数的调用帧，否则就无法进行尾调用优化。

* + - * 1. 尾递归

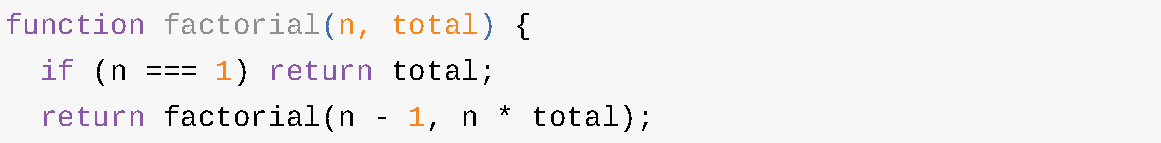
对于尾递归来说，由于只存在一个调用帧，所以永远不会发生栈溢出错误。

* 阶乘

非尾递归（因为返回行还有运算，外层调用帧还要保留变量n）：

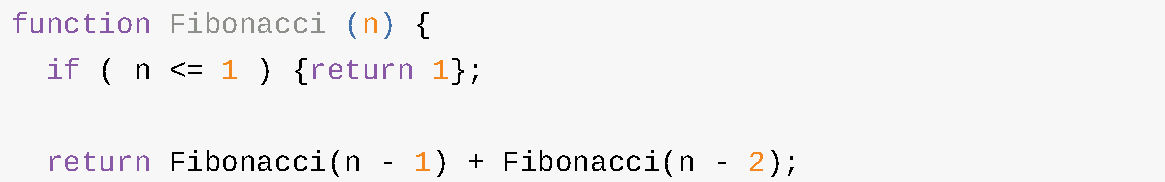


尾递归：

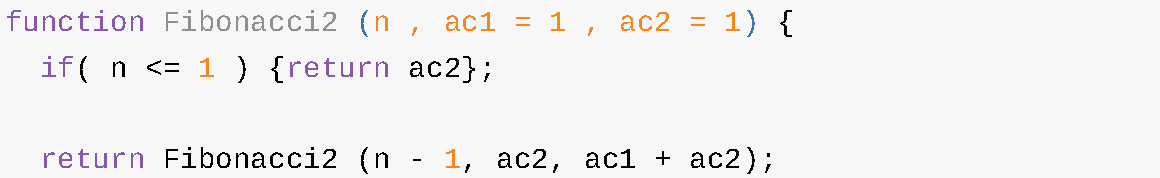


* fibonacci

非尾递归：



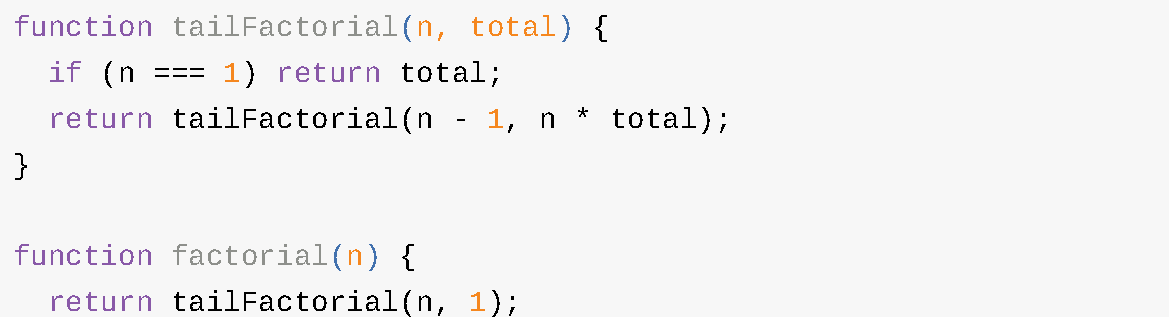
尾递归：



* 递归函数的改写

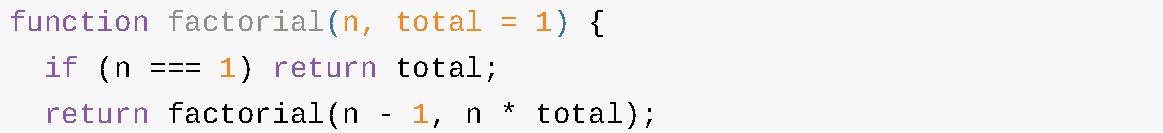
将普通递归改写为尾递归，需要把所有用到的内部变量改写成函数的参数。有两个方法实现。

方法1：在尾递归函数之外，再提供一个正常形式的函数。



函数式编程有一个概念，叫做柯里化，意思是将多参数的函数转换成单参数的形式。这里也可以使用柯里化。

第二种方法就简单多了，就是采用ES6的函数默认值。



ES6的尾调用优化只在严格模式下开启，正常模式是无效的。因为正常模式下，函数内部有两个变量arguments和caller，可以跟踪函数的调用栈。严格模式禁用这两个变量。

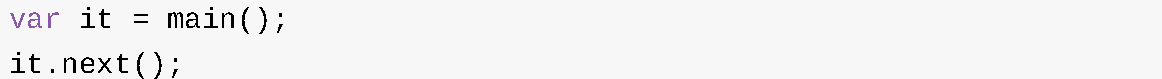
* + 1. 应用
       1. 异步操作的同步化表达

Generator函数的暂停执行效果，意味着可以把异步操作写在yield语句里面，等到调用next方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在yield语句下面，反正要等到调用next方法时再执行。所以，Generator函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

用Generator编写的ajax操作：

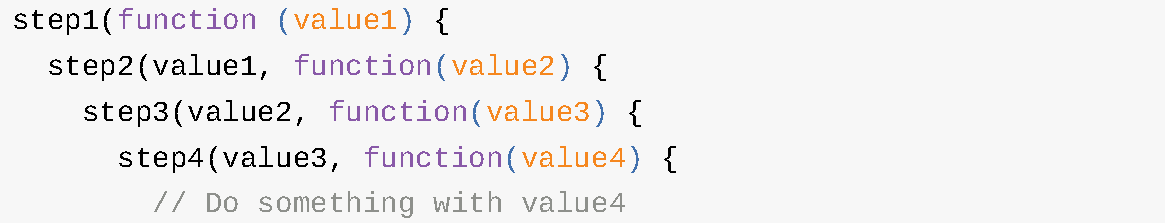




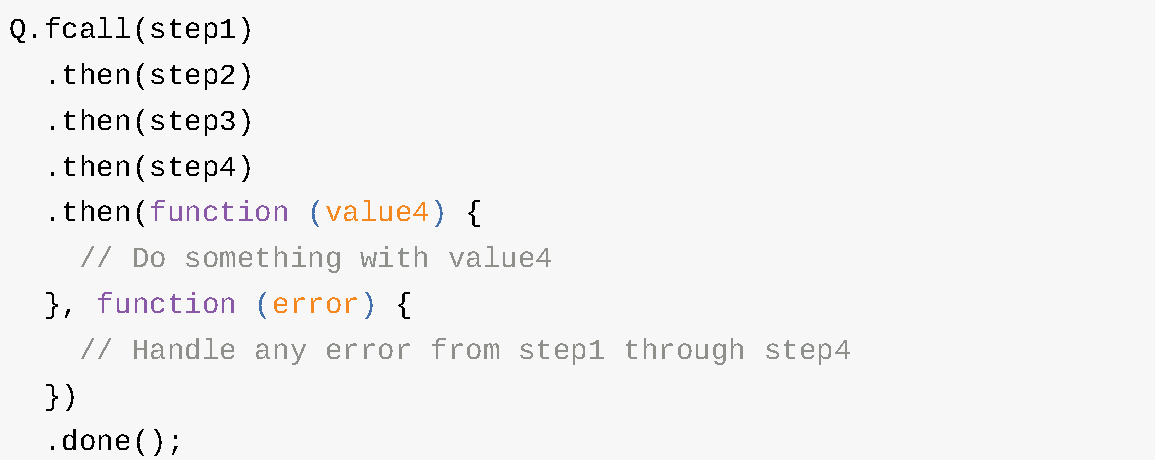


* + - 1. 控制流管理

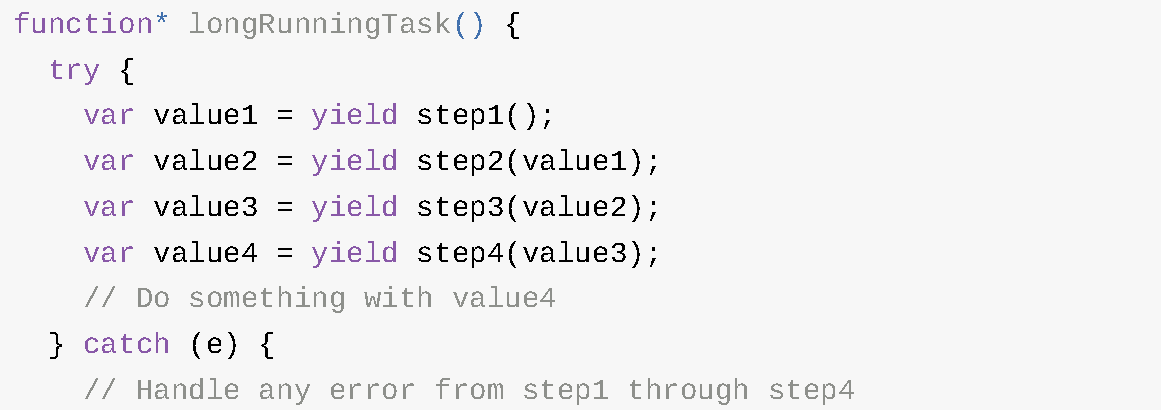
采用回调函数可能为：



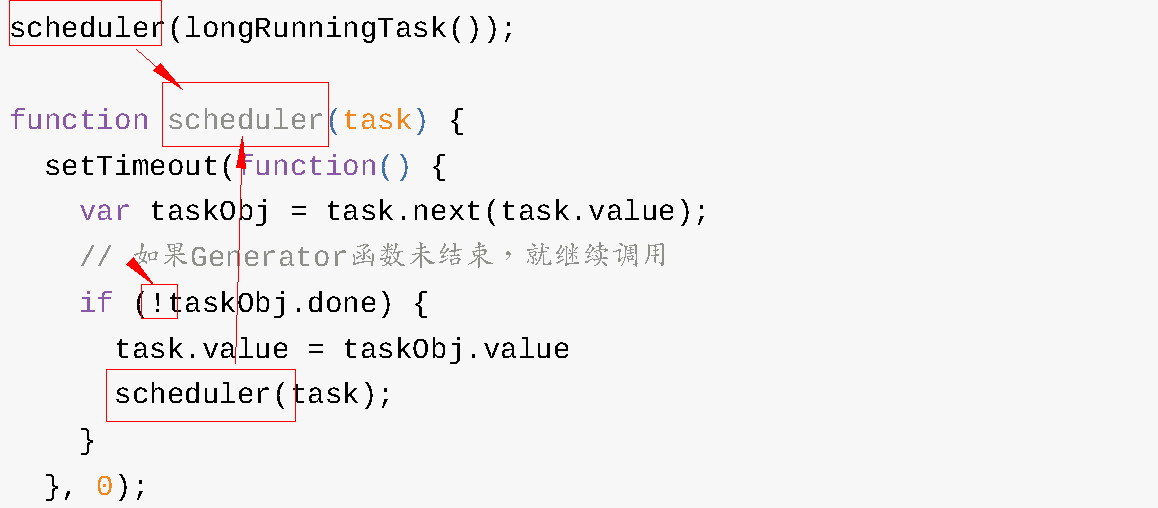
采用Promise可以为：



采用Generator函数可以为：



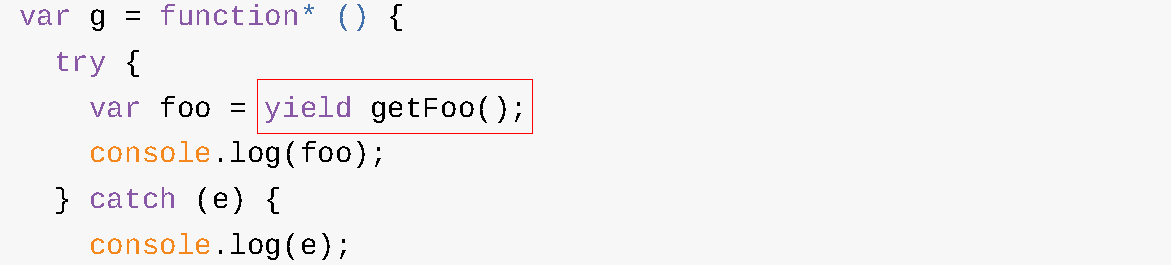
可以使用一个函数，按次序自动执行所有步骤。

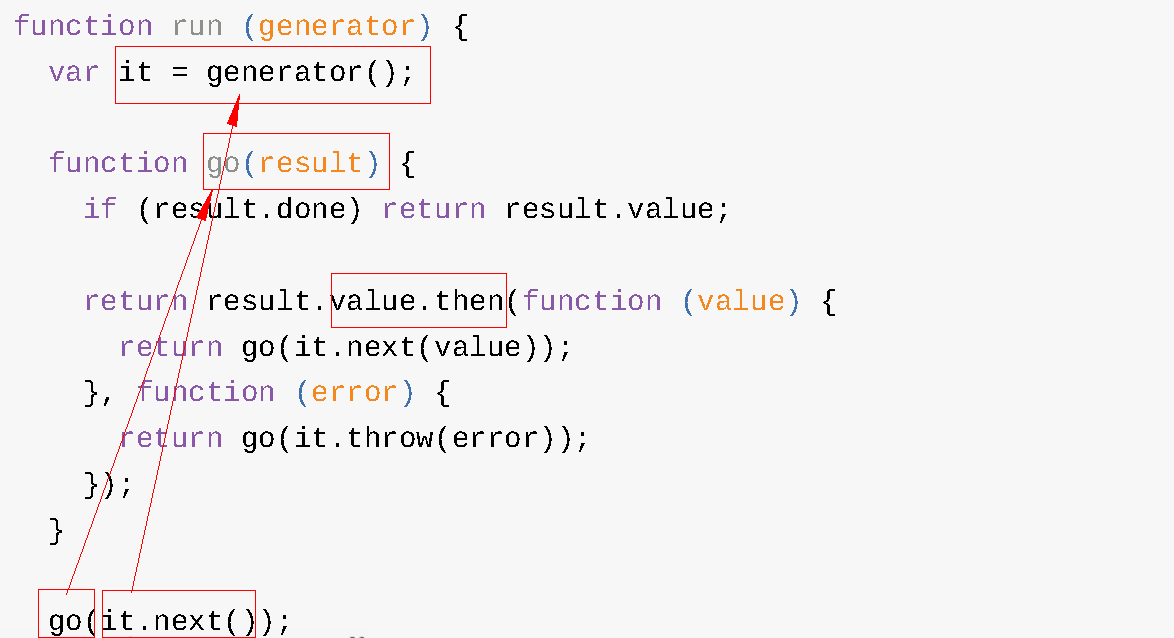


* + 1. Generator函数与Promise的结合

Generator函数与Promise的结合使用Generator函数管理流程，遇到异步操作的时候，通常返回一个Promise对象。









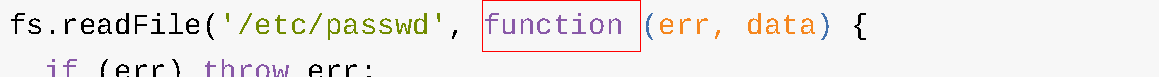
g为Generator函数，yield出Promise对象。run为自动执行Generator函数，先得到遍历器it，然后递归调用内部go方法。如果g结束，就结束。如果还yield出Promise对象，等待promise状态改变后，在promise.then方法中递归调用go方法，执行Generator函数yield出的下一个Promise对象（go方法的返回值没有意义，then方法返回的还是promise，重点在于执行完Generator函数中所有的Promise）。

* + 1. 概念
       1. 异步

简单说就是一个任务分成两段，先执行第一段，然后转而执行其他任务，之后再回过头执行第二段。相应地，连续的执行就叫做同步。

* + - 1. 回调函数

把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。JavaScript语言对异步编程的实现，就是使用回调函数。



* + - 1. Promise

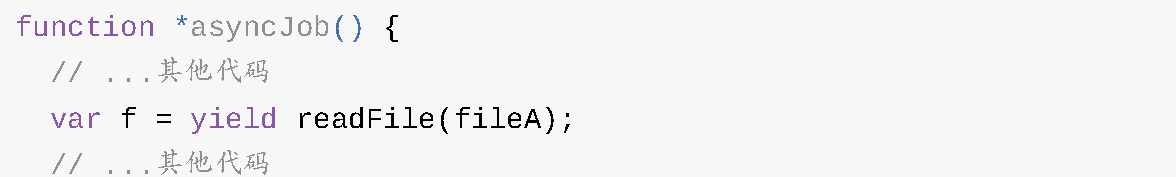
如果存在异步嵌套，使用回调函数会比较混乱。Promise就是为了解决这个问题而提出的，将回调函数的嵌套，改成链式调用。

* + - 1. Generator函数
         1. 协程

多个线程互相协作，完成异步任务。运行流程大致如下：

* 协程A开始执行。
* 协程A执行到一半，进入暂停，执行权转移到协程B。
* 一段时间后，协程B交还执行权。
* 协程A恢复执行。

协程A就是异步任务。协程遇到yield命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。协程的最大优点是代码写法非常像同步操作，如果去除yield命令，简直一模一样。



* + - * 1. Generator函数封装异步

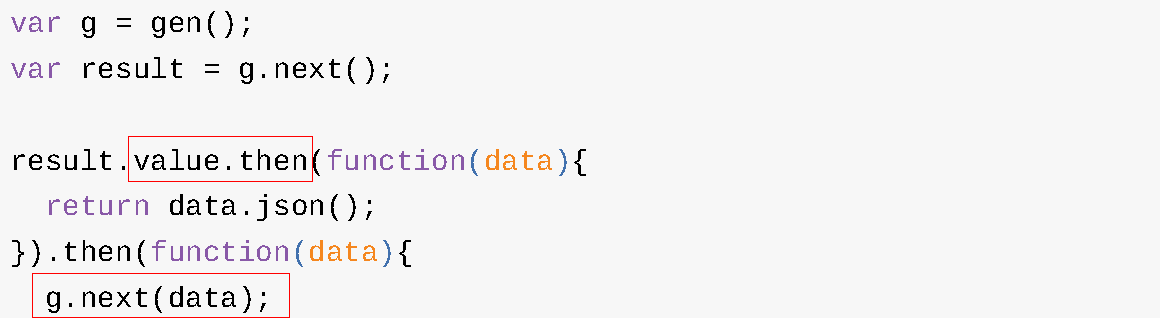
Generator函数就是协程在ES6的实现。1个Generator函数就是对异步任务的封装或容器。异步操作需要暂停的地方，都用yield语句注明。

* + - * 1. Generator函数的数据交换和错误处理

next方法返回值的value属性，用于对外输出数据。next方法接受的参数为向Generator函数体内输入数据。

* + - * 1. Generator函数异步任务的封装





因fetch模块返回的是一个Promise对象，要用then方法调用下一个next方法。

* + 1. Thunk函数
       1. 参数的求值策略

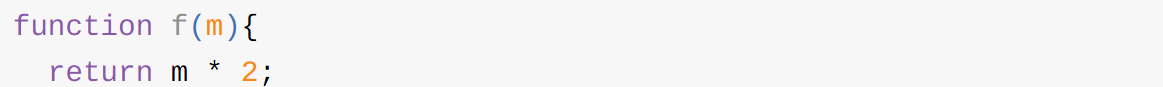
即函数的参数到底应该何时求值。有2种观点：

（1）传值调用：即在进入函数体之前，就计算参数的值，然后再将这个值传入函数。

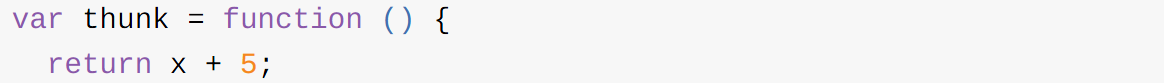
（2）传名调用，即直接将参数的表达式传入函数体，只在用到它的时候求值。

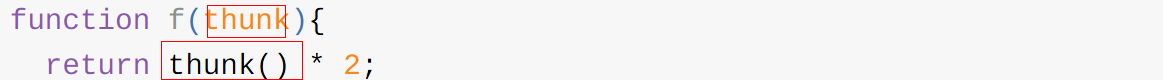
* + - 1. Thunk函数

编译器传名调用的实现，是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做Thunk函数。如：



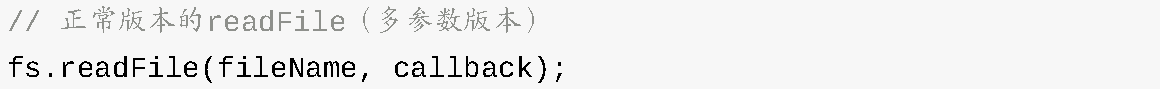






* + - 1. JavaScript语言的Thunk函数

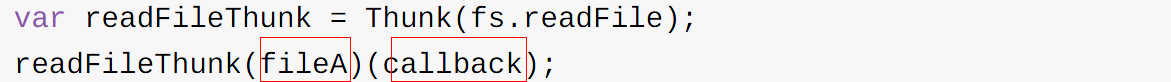
JavaScript语言是传值调用，它的Thunk函数含义有所不同。在JavaScript语言中，Thunk函数替换的不是表达式，而是多参数函数，将其替换成单参数的版本，且只接受回调函数作为参数。





fs模块的readFile方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。这个单参数版本，就叫做Thunk函数。

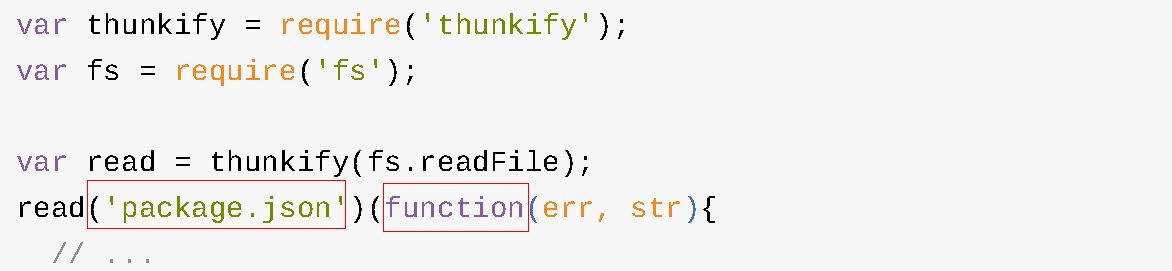
使用如下：



* + - 1. Thunkify模块
* 安装

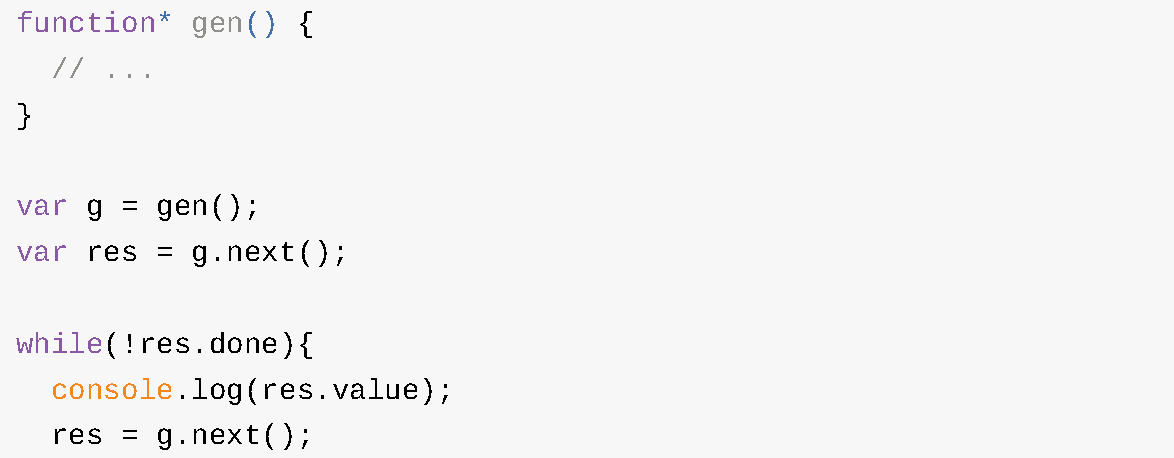


* 使用



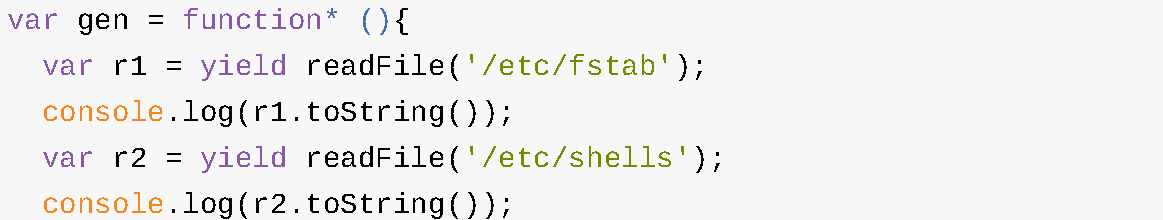
* + - 1. Generator函数的流程管理

可以使用while循环执行完Generator函数。



但是这不适合异步操作。如果必须保证前一步执行完，才能执行后一步，上面的自动执行就不可行。可以使用Thunk函数实现。Thunk函数可用于Generator函数的自动流程管理。





上面的yield命令将程序的执行权移出Generator函数，需要在外部将执行权再交还给Generator函数。

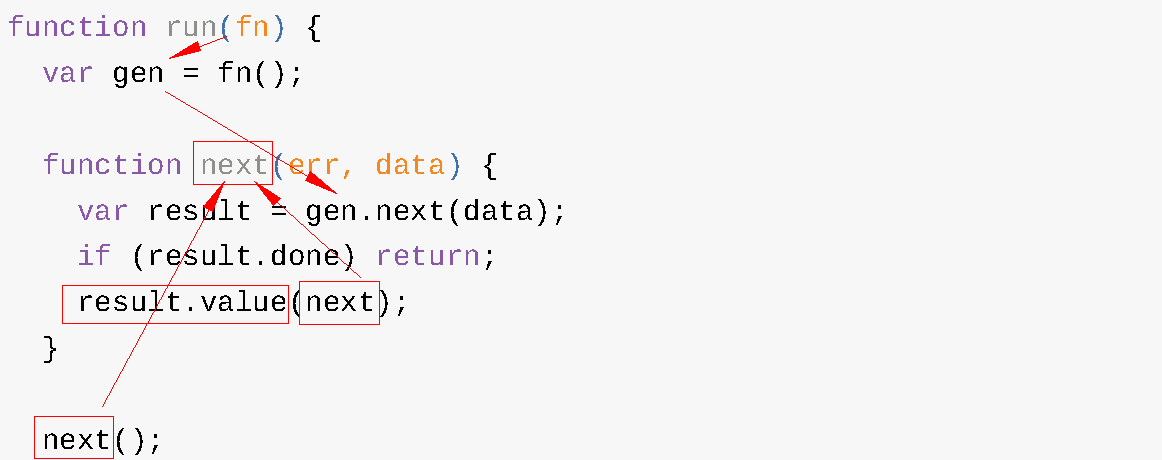
* 手动执行

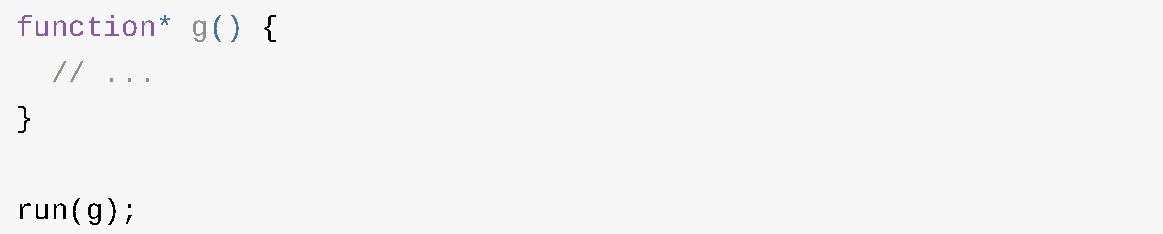
Generator函数的执行过程，就是将同一个回调函数，反复传入next方法的value属性（返回的是readFile函数，回调函数应作为其第2个参数）。



* 自动执行

可以使用递归方法，自动执行Generator函数。

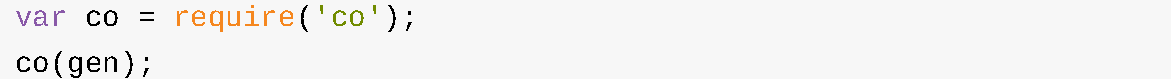


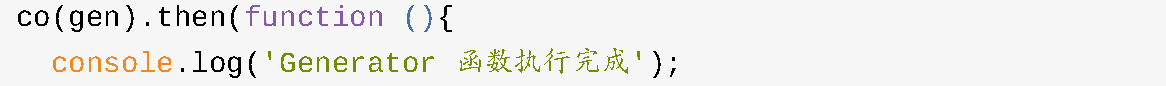


run函数，就是一个Generator函数的自动执行器。内部的next函数就是Thunk的回调函数。如果没结束，就将next函数再传入Thunk函数（result.value属性），否则就直接退出。前提是，跟在yield命令后面的必须是Thunk函数。

* + 1. co模块

co模块可以不用手动编写Generator函数执行器代码。Generator函数只要传入co函数，就会自动执行。co函数返回一个Promise对象，因此可以用then方法添加回调函数。





* + - 1. co模块的原理

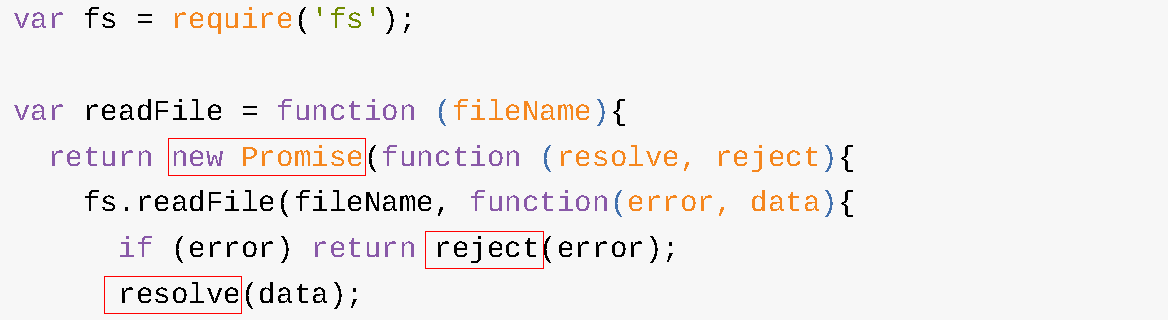
Generator是1个异步操作的容器。其自动执行需要一种机制：即当异步操作有了结果，能够自动交回执行权（执行下一个yield）。有2种方法实现：

* 回调函数：将异步操作包装成Thunk函数，在回调函数里面交回执行权。
* Promise对象：将异步操作包装成Promise对象，用then方法交回执行权。

co模块其实就是将两种自动执行器（Thunk函数和Promise对象），包装成一个模块。使用co的前提条件是，Generator函数的yield命令后面，只能是Thunk函数或Promise对象。

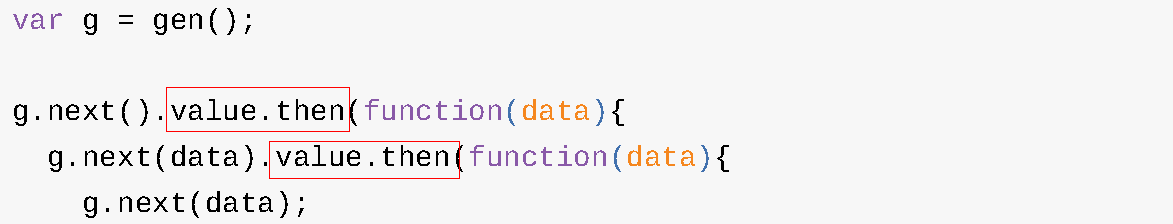
* + - 1. 基于Promise对象的自动执行

首先，把fs模块的readFile方法包装成一个Promise对象。

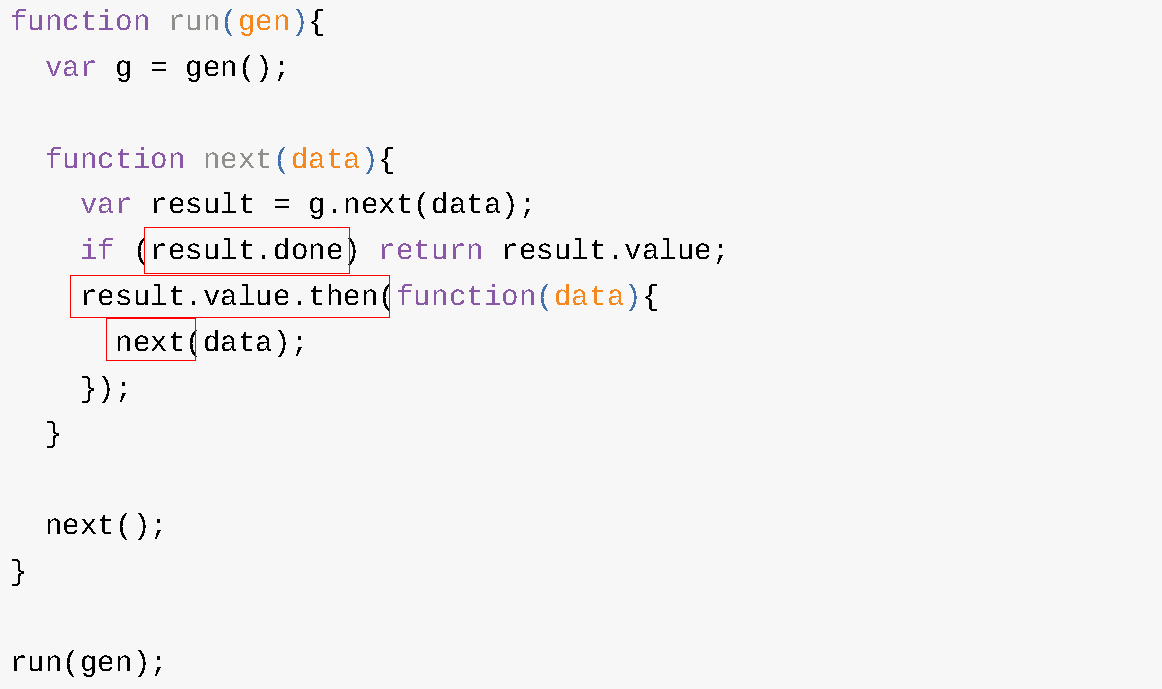




然后，手动执行上面的Generator函数。



手动执行其实就是用then方法，层层添加回调函数。自动执行器（在then中交换执行权）如下：



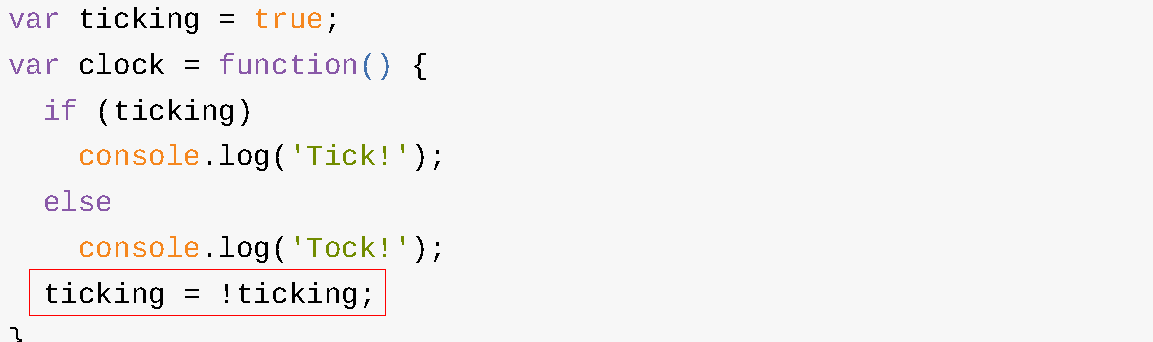
只要Generator函数还没执行到最后一步，next函数就调用自身，以此实现自动执行。

* + - 1. co模块的源码

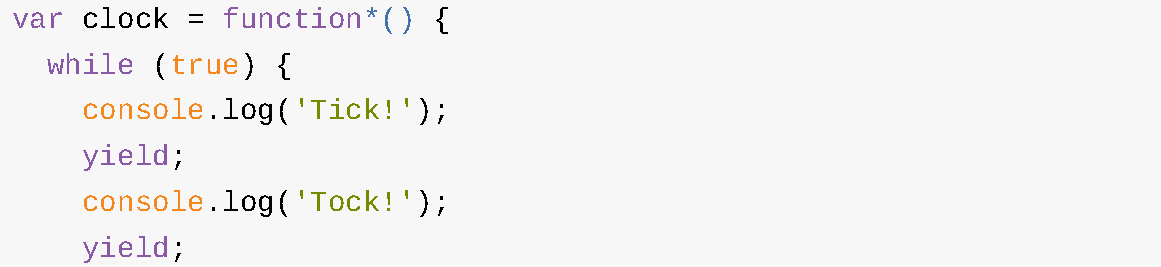
co就是上面那个自动执行器的扩展。

* + 1. Generator与状态机

其他方式实现的状态机：



Generator实现：



Generator之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。